This is a blank page to get the numbering right.

A Simulator for the L Architecture in COMMON LISP

by

Michael R. Blair

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 1986, in partial fulfillment for the degree of Bachelor of Science in Computer Science and Engineering.

Abstract

The object of this paper is to provide a simulator for the Project L architecture. Although the L architecture is designed as a multiprocessor environment, The L Simulator simulates the behavior of a single L processor. This reflects the goals of the initial Simulator, which were twofold:

1. To provide a minimally sufficient operational semantics for the L architecture through which the raw performance of L may be evaluated.
2. To constitute a "sandbox" in which one can easily extend and revise the existing L proposal.

Thus, extending the current L Simulator to simulate multiple concurrent L processors is a casual modification, as are major modifications to the instruction set.

Thesis Supervisor: Stephen A. Ward, Ph.D.

Title: Associate Professor of Computer Science and Engineering

Acknowledgements

I would first like to thank my department advisor Prof. Patrick H. Winston for giving me the opportunity to hack in the MIT Artificial Intelligence Laboratory, providing me with the insight to discover my real interests and the confidence to pursue them. I would further like to thank Prof. Stephen A. Ward, my thesis advisor, for being a mentor, a colleague, a father figure, and a close friend. Also, my gratitude and respect are offered to Prof. Richard Zippel who helped me grapple with the "features" of the Explorer[1] LISP machine system while developing the code in this thesis. Finally, I warmly thank Sharon Thomas for her support (and occasional kick in the pants) during my stay at MIT.

To my parents and family, I offer my appreciation for the moral support I needed in pursuing my education thus far. Through their love and respect they gave me the independence I needed to grow and mature, and they were always there when I needed them (though I was usually never home when they called).

Finally, though perhaps most importantly, I thank The Timken Company Educational Scholarship Fund, Incorporated for providing me with the opportunity to pursue my ambitions. One can never repay such a debt as being given one's future.

[1]Explorer is a trademark of Texas Instruments Inc.

Contents

Chapter 1

Introduction

The goal of this thesis is to provide a flexible simulator for the L Architecture as proposed in [4]. Since it is our goal to eventually implement the L computational model in μcode on an Explorer[1] LISP machine, the Simulator[2] was implemented on this same machine. Since the emphasis on this first generation simulator is on modeling the L functionality in a flexible code framework, I have chosen to implement this first incarnation entirely in COMMON LISP . This reflects our desire to provide a simple uniform environment in which to explore modifications to the L proposal, initially avoiding a complex and difficult to maintain μcode environment. Thus, often the structure of the Simulator will reflect our need for flexibility as it dominates our desire for performance. Further, since are initial goal is merely to capture the functionality of the L instruction set, I have also limited myself to simulating a single L processor rather than a pool of concurrent processors, though throughout the code I have paid careful attention to the details necessary for this multiprocessor extension. As we shall conclude, this extension is quite trivial.

In presenting the details of the L architecture, I have assumed the reader's familiarity with the notion of *chunks*, the data structure on which L is based. A complete introduction may be found in [4], so I will make no effort to duplicate it here. Thus, I will liberally assume the reader's familiarity with the ideas underlying, say, creating a state chunk or chasing the instruction stream through a chain of code chunks. My focus will therefore be on the non-obvious issues involved in executing this instruction set and on the style of implementation of the Simulator. Thus, I begin with a

[1] Explorer is a trademark of Texas Instruments Inc.

[2] Throughout, I will use the capitalized spelling to distinguish the particular implementation of the L Simulator discussed here to distinguish it from simulators in general.

discussion of the L operand addressing modes as implemented on the L Simulator.

Chapter 2

The L Instruction Set

The L instruction set is a simple load/store instruction set with certain extensions for multiprocessor support. We will deal first with the addressing modes supported by L then will proceed with the actual L instructions.

2.1 L Addressing Modes

The most distinguishing feature of L is the notion of a *chunk*. Since the chunk is the sole data object supported by the L architecture — the data object from which all others are constructed — it is not surprising that the addressing modes offered in [7] are centered on the notion of accessing chunk contents. Since our purpose here is to explore the implementation of the L Simulator, and specifically *not* to critique the adequacy of the instruction set or its opcode encodings, I will restrict my discussion to only those issues directly relevant to the topic at hand, reserving comment on future modifications to the final chapter. With that in mind, let's examine the source operand addressing modes.

2.1.1 Instruction Stream Access

A preliminary step to implementing the source operands is to provide a mechanism for accessing a byte from the instruction stream code chunk. This is accomplished by the `next-istream-byte!` function below. This function makes liberal use of the chunk data structure support implemented in [4].[1]

[1]This code, modestly modified and expanded for our use here, appears in Appendix B.

```
;; Grab the next code byte from the instruction stream, resolving chunk-byte references
;;  ... may cross a code chunk boundary.

(defsubst next-istream-byte!()
  (let ((code-chunk-ptr    (PChunk *current-state-chunk*))
        (code-chunk-offset (PCOff  *current-state-chunk*)))

      ;; At any (aligned) istream byte, we may stumble across the continuation
      ;;  pointer ... this entails an implicit jump to the continuation of the code.

      (if (ref-p code-chunk-ptr (ldb (byte 3 2) code-chunk-offset))
          (progn (setf (PChunk *current-state-chunk*)
                       (chunk-elt code-chunk-ptr (ldb (byte 3 2) code-chunk-offset)))
                 (setf (PCOff  *current-state-chunk*) 0)))

      ;; NOTE: pointers to pointers will trap to the error handler by chunk-byte()
      (let ((code-chunk-offset (PCOff  *current-state-chunk*)))
        (prog1 (chunk-byte (PChunk *current-state-chunk*) code-chunk-offset)
               (let ((postinc-uPC (1+ code-chunk-offset)))
                 ;; check for falling off the end of the chunk
                 (if (> postinc-uPC #b11111)
                     (error 'L-RUNTIME-ERROR
                            "NEXT-ISTREAM-BYTE!: postinc(uPC) chunk-boundary overflow"))
                 (setf (PCOff *current-state-chunk*) postinc-uPC))))))
```

This code is fairly straightforward. The first idea is check if the chunk element specified by the three most significant bits of the μPC pointer (the current code chunk element in the instruction stream) is a chunk reference. If it is, we take the implicit jump to that code chunk and set the μPC to 0 (the first byte of the new code chunk). The second idea is to check that we do not "fall off the edge" of the code chunk when incrementing the μPC to the next istream byte. An alternative would be to allow this to roll over to 0, this allowing an implicit branch to the beginning of the current code chunk. However, such a μPC roll-over is much more likely to arise from programmer error than from excruciating byte packing, so signalling an error is preferred. Finally, no more than one level of indirection is allowed in chunk references. Thus, references that in turn resolve to references will be trapped by `chunk-byte`.[2] This is a desirable limitation since we thereby restrict the propagation of side-effects (state-changes) to a unit distance, enabling dynamic analysis of possible parallelism in the multiprocessor case. This idea will be elaborated in a forthcoming memo by Ward. Note that for efficiency in the compiled code, this function has been defined as an in-line substitution routine (a `defsubst`), as have most of the other subroutines in the Simulator. This too will be justified in a later chapter.

2.1.2 Source Operands

Given this mechanism for accessing instruction stream bytes, together with the more mundane chunk access capabilities implemented in Appendix B, the majority of ad-

[2] See Appendix B for details

dressing modes outlined in [7] are comparably straightforward. This reflects our ability in COMMON LISP to define explicit abstraction barriers between the various levels of detail in our system. Such modular design is the trademark of a flexible system. Little more need be said about the remainder of the source operands and their encodings, with the exception of the more `(S ...)` and `(byte ...)` modes which are discussed below. Such is the reward of gratuitously documented code.

```
;;;
;;; Source addressing mode handlers
;;;

;;; (Sn): nth element of current state
;;;
;;;   CODE: <mode-encoding> + n  for n<8  [1 byte]

(defun source-Sn(encoding)
  (chunk-elt *current-state-chunk* (logand encoding 7)))


;;; (Sn i): ith element of chunk referenced by nth element of current state
;;;
;;;   CODE: <mode-encoding> + 8*i + n  for  i<9, n<8   [1 byte]

(defun source-Sn-i(encoding)
  (chunk-elt (chunk-elt *current-state-chunk* (logand encoding 7))
             (1- (lsh encoding -3))))
```

The following two operand modes present somewhat of a problem. We would like to encode these various combinations in a compact set of source codes, but this would come at the price of having to implement a distinct function for each addressing mode. For example, the `(S (byte Sn i) j)` mode would be a distinct mode from the `(S n (byte Sm j))` mode. I have chosen to implement this instead as a very general addressing mode whose arguments `<n>` and `<i>` may consist of any combination of source operand encodings. This greatly simplifies the code through the beauty of recursive calls to `source-operand!`, but does so at the cost of perhaps unwanted flexibility and a bit more runtime overhead. One could justify the unwanted flexibility by offering that such restrictions should be enforced by the assembler or compiler. However, looking ahead to μcode implementations of the Simulator, the argument that these should be distinct source codes predominates. After all, we have source codes to spare (256 total) whereas supporting recursive μcode is a needless inefficiency. Also, allowing arbitrary depth recursion destroys the possibility of meaningful byte counting in the instruction stream. Thus, efforts to prefetch operands would be thwarted by an inability to know, on the basis of the source code, how many operands follow and how to process them. The next generation Simulator will therefore most likely split these two source codes into several distinct codes, but perhaps still providing a special catchall (recursive) mode for flexibility with say a limit on the depth of recursion … or not.

```
;;; (S <n> <i>): <i>th element of chunk referenced by <n>th element of current state
;;;              where  <n> is a constant n or (byte Sn i) which must resolve to a constant < 8
;;;                     <i> is a constant i or (byte Sm j) which must resolve to a constant < 9
;;;        Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <n>, <i>   [4 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <n> and <i> to be arbitrary source modes, but they
;;;              must both resolve according to the above stated constraints else the error
;;;              handler will be invoked by chunk-elt.

(defun source-S-n-i()
  (let ((chunk-ref (chunk-elt *current-state-chunk* (source-operand!))))
    (chunk-elt chunk-ref (source-operand!))))

;;; (byte <chunk-ref> <index>): <index>th byte within the chunk referenced by <chunk-ref>
;;;                                   where  <chunk-ref> may be Sn or (byte Sn i)
;;;                                          <offset>    may be  j or (byte Sm j)
;;;        Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <chunk ref>, <index>   [3 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <chunk-ref> and <offset> to be arbitrary source modes.
;;;              However, <chunk-ref> must resolve to a chunk-ref and <offset> must resolve
;;;              to a constant else the error handler will be invoked.
;;;              [See code for chunk-byte and chunk-elt].

(defun source-byte-c-i()
  (let ((chunk-index (source-operand!)))
    (chunk-byte (chunk-elt *current-state-chunk* chunk-index)
                (source-operand!))))
```

The remainder of the source codes are again relatively straightforward and are therefore presented without further comment.

```
;;; (byte Sn i): ith byte of chunk referenced by Sn. n=8 for current state chunk.
;;;
;;;   CODE: <mode-encoding>, 16*i + n   [2 bytes]

(defun source-byte-Sn-i()
  (let* ((current-code-byte (next-istream-byte!))
         (n (logand current-code-byte #xF)))
    (chunk-byte (if (= n 8)                    ; n=8 means current state chunk
                    *current-state-chunk*
                    (chunk-elt *current-state-chunk* n))       ; else Sn
                (lsh current-code-byte -4))))

;;; S: current state
;;;
;;;   CODE: <mode-encoding>   [1 byte]

(defun source-S() *current-state-chunk*)

;;; k: medium (2 byte) constant
;;;
;;;   CODE: <mode-encoding>, klo, khi   [3 bytes]

(defun source-2-byte-k()
  (+ (next-istream-byte!) (* 256 (next-istream-byte!))))

;;; k  for  -64 <= k < 64: 7 bit 2's complement constant.
;;;
;;;   CODE: 128 + k   [1 byte]

(defun encode-source-1-byte-k()
  (setq *current-L-sourcecode* 128)
  (loop for i from 0 below 128
        do (eval `(defsourcecode
                    ,(if (> i 63)                             ; Negative 2's complement?
                         (- (logand #x3F (1+ (lognot i))))    ; 1's complement, increment,
                                                              ;    mask, and invert
                         i)))))
```

2.1.3 Destination Operands

Most of the destination operands are analogous to the source operands with certain obvious exceptions (for instance we disallow storing into the instruction stream). For consistency, the `(S ...)` and `(byte ...)` modes are implemented in a generalized (recursive) fashion. The only substantial differences between the source and destination implementations are the additional modes for modify the Prerequisite bits. In the current implementation, prerequisites are not fully implemented since a satisfactory strategy for distributing these bits has not been fully explored at the date of publication. This will be a topic of further research. One final note is the curious way in which destination codes return multiple values. This is because one need return to the caller an indication of whether the denoted destination constitutes a byte of a full word (4 bytes) with the specification of the target chunk and target item (byte or full slot) within that chunk. The reason for this when we do the actual store, we use the `setf` function of Common Lisp . Unfortunately, since Common Lisp does not support the notion of a datum address (*i.e.* "pointers"), we must return the necessary information for the caller to do the store. An alternative would be to distribute the stores within these destination functions. However, this would entail passing the value of the source datum to the destination decoding functions. I have opted for the multiple value strategy for simplicity and and coherence. Following, therefore, are the destination operand handlers.

```
;;;
;;; Destination addressing mode handlers
;;;

(defconstant SLOT-DEST   T)                    ; To be used as boolean flags
(defconstant BYTE-DEST NIL)                    ; ...  see code for store-dest()

;;; Following N destination codes are analogous to the source encodings.
;;; Note that these routines return multiple values, from which storedest()
;;;  can construct the destination address.

;;; (Sn): nth element of current state.
;;;        Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding> + n  for  n<8   [1 byte]

(defun dest-Sn(encoding)
  (values SLOT-DEST *current-state-chunk* (logand encoding 7)))


;;; (Sn i): ith element of chunk referenced by nth element of current state.
;;;        Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding> + 8*i + n  for  i<9, n<8   [1 byte]

(defun dest-Sn-i(encoding)
  (values SLOT-DEST (chunk-elt *current-state-chunk* (logand encoding 7))
                  (1- (lsh encoding -3))))

;;; (S <n> <i>): <i>th element of chunk referenced by <n>th element of current state
;;;              where  <n> is a constant n or (byte Sn i) which must resolve to a constant < 8
;;;                     <i> is a constant i or (byte Sm j) which must resolve to a constant < 9
;;;        Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <n>, <i>   [4 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <n> and <i> to be arbitrarily defined modes, but they
;;;              must both resolve according to the above stated constraints else the error
;;;              handler will be invoked by chunk-elt.

(defun dest-S-n-i()
  (let ((chunk-ref (chunk-elt *current-state-chunk* (source-operand!))))
    (values SLOT-DEST chunk-ref (source-operand!))))
```

```
;;; (byte <chunk-ref> <index>): <index>th byte within the chunk referenced by <chunk-ref>
;;;                                   where  <chunk-ref> may be Sn or (byte Sn i)
;;;                                          <offset>    may be  j or (byte Sm j)
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <chunk ref>, <index>   [3 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <chunk-ref> and <offset> to be arbitrary dest modes.
;;;               However, <chunk-ref> must resolve to a chunk-ref and <offset> must resolve
;;;               to a constant else the error handler will be invoked.
;;;               [See code for chunk-byte and chunk-elt].

(defun dest-byte-c-i()
  (let ((chunk-index (source-operand!)))
    (values BYTE-DEST
            (chunk-elt *current-state-chunk* chunk-index)
            (source-operand!))))

;;; (byte Sn i): ith byte of chunk referenced by Sn. n=8 for current state chunk.
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, 16*i + n   [2 bytes]

(defun dest-byte-Sn-i()
  (let* ((current-code-byte (next-istream-byte!))
         (n (logand current-code-byte #xF)))
    (values BYTE-DEST
            (if (= n 8)                 ; n=8 means current state chunk
                *current-state-chunk*
                (chunk-elt *current-state-chunk* n))    ; else Sn
            (lsh current-code-byte -4))))

;;; (R j Sn): nth element of current state.
;;;         Sets Sn as READ-ONLY prerequisite j.
;;;
;;;   CODE: <mode-encoding> + 8*j + n   for  j<4, n<8   [1 byte]

(defun dest-R-j-Sn(encoding)
  (let ((target-slot (ldb (byte 3 0) encoding)))
    (prog1 (values SLOT-DEST encoding target-slot)
           (setf (RPreq-bit *current-state-chunk* target-slot) (ldb (byte 2 3) encoding)))))

;;; (W j Sn): nth element of current state.
;;;         Sets Sn as READ/WRITE prerequisite j.
;;;
;;;   CODE: <mode-encoding> + 8*j + n   for  j<4, n<8   [1 byte]

(defun dest-W-j-Sn(encoding)
  (let ((target-slot (ldb (byte 3 0) encoding)))
    (prog1 (values SLOT-DEST encoding target-slot)
           (setf (WPreq-bit *current-state-chunk* target-slot) (ldb (byte 2 3) encoding)))))

;;; (N j Sn): nth element of current state.
;;;         CLEARS prerequisite j.
;;;
;;;   CODE: <mode-encoding> + 8*j + n   for  j<4, n<8   [1 byte]

(defun dest-N-j-Sn(encoding)
  (let ((target-slot (ldb (byte 3 0) encoding)))
    (prog1 (values SLOT-DEST encoding target-slot)
           (setf (RPreq-bit *current-state-chunk* target-slot) 0)
           (setf (WPreq-bit *current-state-chunk* target-slot) 0))))
```

2.2 L Operations

Having established the addressing mode functionality, let us accept on faith that we are given subroutines `source-operand!` and `destination-operand!` which decode the next source/destination operand into its appropriate source datum/destination information respectively, adjusting the μPC to always point to the next appropriate byte in the instruction stream.[3] The instruction set can therefore build nicely on the layer of abstraction provided by these two subroutines. The majority of the implementation is therefore relatively uneventful except for the few twists noted below. Do note, however, the nature of the return values of each instruction. These will facilitate an informative single stepper for the user interface. We begin with the simple arithmetic/logic instructions.

```
;;; Nullary opcode

(defun NOP() 'NOP)                                    ; NOP ... returns 'NOP

;;; Arithmetic and Logic Instructions

(defun L-add(x y)                             ; ADD: 32-bit 2's complement ... sets CCs
  ;; V - Overflow bit: arguments of same sign, but result different ... toggle sign of result
  ;; N - Negative bit ... toggled upon overflow
  ;; C - Carry bit: carry from Most Significant Bit
  ;; Z - Zero bit
  (let* ((sign-x (logand #x80000000 x))
         (sum (+ x y))
         (result (logand #xFFFFFFFF sum))
         (sign-result (logand #x80000000 result))
         (CCs (+ (if (zerop result) Z-BIT 0)
                 (if (zerop sign-result) 0 N-bit)
                 (if (zerop (logand #x100000000 sum)) 0 C-BIT))))

    (if (and (eq sign-x (logand #x80000000 y))        ; arguments have like signs ...
             (not (eq sign-result sign-x)))           ; ... but differ from result's sign
        (set-CCs (logxor CCs (+ N-BIT V-BIT)))        ; ... so force V and toggle N-bit
        (set-CCs CCs))                                ; Install the CC bits
    result))
```

[3]These two functions will be precisely defined when discussing the structure of the Simulator.

```
(defun L-sub(x y)                              ; SUBTRACT: 32-bit 2's complement ... sets CCs
  ;; V - Overflow bit: args' signs differ as do signs of x and result ... toggle sign of result
  ;; N - Negative bit ... toggled upon overflow
  ;; C - Carry bit
  ;; Z - Zero bit
  (let* ((sign-x (logand #x80000000 x))
         (x-same-sign-as-y? (eq sign-x (logand #x80000000 y)))
         (diff (- x y))
         (result (logand diff #xFFFFFFFF))
         (sign-result (logand #x80000000 result))
         (carry (logand #x100000000 diff))
         (x-and-result-signs-differ? (not (eq sign-result sign-x)))
         (CCs (+ (if (zerop result) Z-BIT 0)
                 (if (zerop sign-result) 0 N-bit)
                 carry)))

    (if (and x-same-sign-as-y?                 ; arguments have like signs ...
             x-and-result-signs-differ?)       ; ... but differ from result's sign ==> C
        (setq CCs (+ CCs C-BIT)))              ; ... so force a C bit
    (if (and (not x-same-sign-as-y?)           ; arguments have different signs ...
             x-and-result-signs-differ?)       ; ... and signs of x and result differ
        (set-CCs (logxor CCs (+ N-BIT V-BIT))) ; ... so force V and toggle N
        (set-CCs CCs))                         ; Install the CC bits
    result))

(defun L-neg(x)                                ; NEGATE: 32-bit 2's complement ... sets CCs
  (L-sub 0 x))                                 ; Special case of L-SUB

(defun L-move(x) (L-add x 0))                  ; return the source operand, setting CCs.


;;; Compare: x-y ... set CCs only (i.e. no destination)

(defun L-compare()
  (let* ((x (source-operand!))
         (y (source-operand!)))
    (L-sub x y)))              ; No store-dest.
```

Notice the untraditional treatment of inter-routine relationships. Traditionally one would define `subtract` and `negate` then define `add` in terms of these.[4] However, this is complicated by noting that the negate may itself cause an overflow; thus condition codes must be inspected after the negate as well as after the add. As an alternative, I have implemented `add` and `subtract` directly, setting the condition codes as per convention established in the *PDP*11[5] processor family.[6] Thus, negate may be defined as a subtract from condition codes, again as per the *PDP*) while `compare` is a subtract with no store into a destination. The logical operations are simple in comparison.

[4] *e.g.* `(add x y) = (sub x -y)`.

[5] *PDP* is a trademark of Digital Equipment Corp.

[6] See [2].

```
;;; Logic Opcodes ... set CCs

(defun L-and() #'logand)                        ; Use some LISP primitives ...
(defun L-or () #'logior)

(defun L-not(x) (L-add (lognot x) 0))           ; 1's complement
```

Jumps are notably more clever. A jump may be a jump into another code chunk (a long jump, or simply *jump*) or it may be a jump within the current code chunk (a short jump, or *branch*). The long jump will therefore require a full word (32 bit) chunk reference whereas the branch need only specify the target byte within the current code chunk. As explained below, we accomplish both styles with a single opcode followed by a so-called *jumpcode*, an idea spawned by Alan Yueh. See the code for details.

```
;;; Unconditional Jumps (the long and short of it)

;;; Jump opcodes are followed by a jump-code, where a jump-code is of the format:
;;;
;;;     branch/jump     jump-code     side-effect(s)
;;;     -----------     ---------     --------------
;;;
;;;       Branch:        000sssbb      uPC := sssbb (sss = slot; bb = byte within the slot)
;;;
;;;         Jump:        PPPsssbb       PC := contents of slot PPP of current code chunk
;;;                     (PPP ~= 000)   uPC := sssbb
;;;
;;;
;;;     ... Clearly then, a branch is just a short jump, a jump is just a long branch,
;;;     and a full continuation is a hop, skip and a jump.
;;;    (And a horse is a horse, of course, of course.)

(defun L-jmp()
  (let ((jump-code-byte (next-istream-byte!)))
    (let ((branch-if-zero-else-target-slot (ldb (byte 3 5) jump-code-byte))
          (target-uPC (mask-field (byte 5 0) jump-code-byte)))
      (if (not (zerop branch-if-zero-else-target-slot)) ; jump?
          (setf (PChunk *current-state-chunk*)
                (chunk-elt (PChunk *current-state-chunk*) branch-if-zero-else-target-slot)))
      (setf (PCOff *current-state-chunk*) target-uPC)   ; branch

      ;; Informative return value for single stepping ...
      (if (zerop branch-if-zero-else-target-slot)
          'BRANCH
          (if (zerop (PCOff *current-state-chunk*))
              'JUMP
              'JUMP&BRANCH)))))
```

Building on this abstraction, we may build any variety of conditional jump. Note that each conditional will follow the same format as the `jmp`, *viz.* an opcode followed by a *jumpcode* byte. Thus, all conditionals may be branches or full jumps by building on the implementation of `jmp`. The style of code here exploits the common functionality among conditionals. This is an adaptation of earlier work done in the μcode in [6], a style inspired by Ward.

```
;;; Conditional Jumps ... simply test condition bits then do L-jmp if satisfied,
;;;                           else skip next istream-byte.
```

```
(defsubst jump-iff-0(bits-set?)
  (if bits-set?
      (progn (next-istream-byte!) 'SKIP)       ; If not zero, punt the jump (i.e. skip)
      (L-jmp)))                                ; Else jump

(defsubst jump-iff-not-0(bits-set?)
  (if bits-set?
      (L-jmp)                                  ; If not zero, then jump
      (progn (next-istream-byte!) 'SKIP)))     ; Else punt the jump (i.e. skip)


(defmacro defcondition(jump-condition cc-mask)
  `(,jump-condition (ldb-test (byte 6 0)
                              (logand ,cc-mask (L-CCs *current-state-chunk*)))))


(defun le () (defcondition jump-iff-not-0 Z-BIT))                       ; JE  = JV
(defun lne() (defcondition jump-iff-0     Z-BIT))                       ; JNE = JNV

(defun jle() (defcondition jump-iff-not-0 (logior Z-BIT N-BIT)))        ; JLE
(defun lgt() (defcondition jump-iff-0     (logior Z-BIT N-BIT)))        ; JGT

(defun jlt() (defcondition jump-iff-not-0 N-BIT))                       ; JLT = JN
(defun jge() (defcondition jump-iff-0     N-BIT))                       ; JGE = JNN

(defun jlos() (defcondition jump-iff-not-0 (logior Z-BIT C-BIT)))       ; JLOS
(defun lhi () (defcondition jump-iff-0     (logior Z-BIT C-BIT)))       ; JHI

(defun jlo () (defcondition jump-iff-not-0 C-BIT))                      ; JLO  = JC
(defun jhis() (defcondition jump-iff-0     C-BIT))                      ; JHIS = JNC

(defun jz () (defcondition jump-iff-not-0 Z-BIT))                       ; JZ
(defun jnz() (defcondition jump-iff-0     Z-BIT))                       ; JNZ

(defun jr () (defcondition jump-iff-not-0 R-BIT))                       ; JR
(defun jnr() (defcondition jump-iff-0     R-BIT))                       ; JNR
```

The procedure linkage code again cascades the common functionality among the routines as a means of building abstraction barriers. This makes the code very maintainable since major concepts are isolated from one another. Thus, modifications of code may be concentrated in isolated pockets of code then automatically propagated outward by recompilation.[7] This is an important feature of modular programming, a point well made in [1].

```
;;; Procedure Linkage

(defun L-fork()
  (let ((new-state (new-state-chunk)))
    (setf (ReturnLink new-state) *current-state-chunk*)
    (setf (PChunk new-state) (PChunk *current-state-chunk*))    ; to access the next istream byte
    (setf (PCOff new-state) (PCOff *current-state-chunk*))      ; for the jump ...
    (next-istream-byte!) ; return to byte past the jump code byte
    (setq *current-state-chunk* new-state)
    (L-jmp)
    (activate-state-chunk *current-state-chunk*))
  'FORK)
```

[7] Note that when running interpreted these effects propagate instantly!

```
(defun L-call()
  (suspend-state-chunk *current-state-chunk*)
  (L-fork)
  'CALL)

(defun L-return()
  (let ((return-state-chunk (ReturnLink *current-state-chunk*)))
    ( suspend-state-chunk *current-state-chunk*)
    (activate-state-chunk return-state-chunk)
    (setq *current-state-chunk* return-state-chunk))
  'RETURN)


;;; Thread Control

(defun L-wakeup () (activate-state-chunk (source-operand!)) 'WAKEUP)
(defun L-suspend() ( suspend-state-chunk (source-operand!)) 'SUSPEND)

(defun L-cocall()
  (suspend-state-chunk *current-state-chunk*)
  (L-wakeup)
  'COCALL)

(defun L-thread()
  (L-move (new-state-chunk))
  'THREAD)
```

The following last instructions are not currently implemented. For a single processor simulator, they would be of limited use anyway. However, the next phase of this project will involve researching these issues of implementing locks and keys and prerequisite management. For now, we simply implement them as so-called *scaffolding*.

```
;;; Locks and Keys

(defun L-lock  () (print "UNLOCK: scaffolding") 'UNLOCK)
(defun L-unlock() (print "UNLOCK: scaffolding") 'UNLOCK)


;;; Prerequisite Management

(defun L-preq   () (print   "PREQ: scaffolding")    'PREQ)
(defun L-nopreq () (print "NOPREQ: scaffolding")  'NOPREQ)
(defun L-clrpreq() (print "NOPREQ: scaffolding") 'CLRPREQ)
```

2.3 Summary of Addressing and Instructions

In retrospect, we have herein accumulated all the functional modules that together constitute the L architecture. Our task now is to map these functions onto opcode encodings in an effortless and flexible manner. We also must provide at least a minimal user interface into the code environment. These are the topics of the next chapter which discusses the structure of the L Simulator as a software system.

Chapter 3

Structure of the L Simulator

Having established a collection of functions which constitute an operational semantics for the L Language, we now present the structure of the simulator, which ties these functions together into a software system.

3.1 Language Encoding/Decoding

As with any simulator, the goal is to assign a unique encoding to each instruction, source and destination mode. These encodings are largely arbitrary so we will not dwell on the particular choices presented here. The important issue is that these encodings be assigned in a flexible, automatic way so that modifications and extensions to the language may be made casually. Further, the process of in turn decoding these should be likewise automatic but, moreover, efficient. We therefore adopt a style of generating "aliases" for each defined function. For instance, if we desired the `add` instruction to have encoding 8, we would alias `add` as `L-OPCODE-08`. We may therefore automatically generate a dispatch table of defined handlers, one for each instruction, source and destination mode. Thus, decoding instruction stream bytes can reduce to a simple dispatch through the appropriate dispatch table where the handlers are stored according to their alias [See below]. The following code for establishing source encodings illustrates these ideas. When modifications are made to the language or when encodings are altered, one need merely reconstruct the affected dispatch tables using the function provided below. Similar code is used for encoding destination modes and instructions. See Appendix A for the complete listing.

```
;;;
;;;  Generate uniform sourcecode names and Intern these standardized names
```

```
;;;     for the dispatch table
;;;

(defun L-sourcecode-name(n)
  (intern (format NIL "L-SOURCECODE-~2,'0X" n) 'L))

;;; postincrement(*current-L-sourcecode*)

(defun next-L-sourcecode!()
  (let ((it *current-L-sourcecode*))
    (incf *current-L-sourcecode*)
    it))

;;; defsourcecode(BODY) ==> (defun L-SOURCECODE-xx (sourcecode) BODY)
;;;                                where  xx = postinc(*current-L-sourcecode*)

(defmacro defsourcecode(&body BODY)
  `(defun ,(L-sourcecode-name (next-L-sourcecode!)) (sourcecode) (ignore sourcecode) ,@BODY))

;;; defsourcecodes(startcode no-of-encodings-plus-1 BODY)
;;;      Encodes BODY as N different sourcecodes starting at startcode
;;;          where N := (-1+ no-of-encodings-plus-1) ... i.e. no-of-encodings

(defsubst defsourcecodes(startcode no-of-encodings-plus-1 BODY)
  (setq *current-L-sourcecode* startcode)
  (let ((last-code-plus-1 (+ startcode no-of-encodings-plus-1)))
    (loop while (< *current-L-sourcecode* last-code-plus-1)
          do (eval `(defsourcecode (,BODY ,*current-L-sourcecode*))))))

;;;
;;; Now establish span of various sourcecodes
;;;

(defsubst encode-source-Sn(startcode)
  (defsourcecodes startcode 8 #'source-Sn))

(defsubst encode-source-Sn-i(startcode)
  (defsourcecodes startcode 80 #'source-Sn-i))

;;;
;;; Source operand encodings
;;;

(defun Setup-sourcecode-encodings()

  (encode-source-Sn 0)                          ; SOURCECODES 00-07
  (encode-source-Sn-i 8)                        ; SOURCECODES 08-79 (#x08-#x4F)

  (compiler-let ((*current-L-sourcecode* 123)) nil ; compiler evals first form twice, so pad
    (defsourcecode (source-S-n-i))              ; SOURCECODE 123 (#x7B)
    (defsourcecode (source-byte-c-i))           ; SOURCECODE 124 (#x7C)
    (defsourcecode (source-byte-Sn-i))          ; SOURCECODE 125 (#x7D)
    (defsourcecode (source-S))                  ; SOURCECODE 126 (#x7E)
    (defsourcecode (source-2-byte-k))           ; SOURCECODE 127 (#x7F)
  )

  (encode-source-1-byte-k)                      ; SOURCECODES 128-256 (#x80 - #xFF)

  (values))
```

```
;;;
;;; Source operand dispatch table construction
;;;

;;; Dispatch table for L source addressing mode encodings:

(defvar *L-sourcecode-dispatch-table* (make-array 256))

;;; Handle an illegal sourcecode:

(defun L-sourcecode-illegal(sourcecode)
  (error 'L-RUNTIME-ERROR "Illegal L source operand code: #x~X" sourcecode))

;;; Call this table whenever the definition of any L sourcecode handler

(defun Setup-L-sourcecode-dispatch-table()
  (loop for i from 0 below 256
        do (let ((it (L-sourcecode-name i)))
             (setf (aref *L-sourcecode-dispatch-table* i)
                   (if (fboundp it) (fsymeval it)
                       #'L-sourcecode-illegal)))))

;;;
;;; Dispatch on source operand encoding
;;;

(defsubst source-operand!()
  (let ((sourcecode (next-istream-byte!)))
    (funcall (aref *L-sourcecode-dispatch-table* sourcecode) sourcecode)))
```

Thus, whenever a modification is made to the L Language, the Simulator may be updated automatically by invoking say the following piece of code.

```
;;;
;;;  Set up the SIMULATOR STATE
;;;

(defun Setup-L-simulator()
  (format T     "~&Instruction encodings ............")
  (Setup-instruction-encodings)
  (format T "DONE~%Source mode encodings ............")
  (Setup-sourcecode-encodings)
  (format T "DONE~%Destination mode encodings ........")
  (Setup-destcode-encodings)
  (format T "DONE~%Instruction dispatch table ........")
  (Setup-L-instruction-dispatch-table)
  (format T "DONE~%Source mode dispatch table ........")
  (Setup-L-sourcecode-dispatch-table)
  (format T "DONE~%Destination mode dispatch table ...")
  (Setup-L-destcode-dispatch-table)
  (format T "DONE~%")
  'DONE)

(defun Setup() (Setup-L-simulator))            ; Shorthand
```

In retrospect, using an external data structure (*viz.* the dispatch table) to separate the details of the actual choice of encodings from the simulator code allows the flexibility and maintainability we desire. Further, since any encoding may be fully decoded by a single table dispatch, we achieve our efficiency goals with a blazing **constant time decoding procedure!**

3.2 The Simulator Interface

Once the dispatch tables for source and destination modes and instructions are established, we may again build on this abstraction to define the user interface into the L Simulator. Note that this interface is at present excruciatingly primitive. This will be one area for future development, but for now it is adequate for testing the functionality of L, both by single stepping and via continuous runs. As a matter of taste, when single stepping the Simulator the result of each instruction executed is printed along with the resulting state. This result display is why we were so meticulous with return values in our instructions. For instance, if a branch is taken as a result of a conditional, the Simulator would print the resulting state and `BRANCH`. Further, note that the Simulation begins with the state chunk specified by the global variable `*current-state-chunk*` which must be established prior to the call to LSim. The simulation halts when the current state is suspended, printing the final state and a raw instruction count.

```
(defun LSim(&key (debug? NIL) (stream standard-output))
  (activate-state-chunk *current-state-chunk*)
  (let ((L-instruction-count 0))
    (if debug?
        (loop until (eql SUSPENDED (L-Status *current-state-chunk*))
              do (let ((result (next-step!) ))
                   (incf L-instruction-count)
                   (format stream "~&*current-state-chunk*: ")
                   (LSim-print-chunk *current-state-chunk* stream)
                   (format stream "~&L-Instruction #~D: RESULT = ~A~%" L-instruction-count
                                                                       result)
                   (read-line)))
        (loop until (eql SUSPENDED (L-Status *current-state-chunk*))
              do (progn (incf L-instruction-count) (next-step!))))
    (format stream "~2&>>>>>>><<<<<<<~%>>>  DONE  <<<~%>>>>>>><<<<<<<")
    (format stream "~2% *current-state-chunk* = ")
    (LSim-print-chunk *current-state-chunk*)
    (format stream "~%  ... in ~D L-instructions~2%" L-instruction-count))
  (values))                                   ; Don't return nuttin'
```

Clearly this is a minimally adequate mechanism for testing the Simulator's correctness while also providing a very rough means for evaluating the L language by primitive instruction counting. Evaluation of the efficiency of the Simulator is offered in the following chapter, along with discussion of future extensions and modifications.

Chapter 4

Evaluation of the Simulator

4.1 Performance

The current implementation of the L Simulator executes approximately 100 L-instructions per second. Note that a trade-off was made in favor of execution speed by gratuitous use of macros and in-line code. While this makes little difference running interpreted code,[1] it improves runtime performance of compiled code by avoiding the overhead of function calls, but at notable cost in space. This 100 instruction per second rate is adequate for modest simulations but is uncomfortably slow for substantial simulations. For this reason we intend to implement the major bottlenecks (such as the chunk data structure access) in μcode. This alone should improve performance by a substantial factor. Eventually, the majority of the Simulator will be instantiated in μcode, leaving only the most volatile features in the high level language to facilitate development. This can be used to justify our use of in-line coding to avoid funcall overhead, since by using μcode we will be in effect in-line coding much of the Simulator. This was one of the motivating factors in choosing the Explorer[2] as the development medium for L: Explorers offer lots of accessibility to hacking lots of μcode.

[1] The `defsubst` and `defun` forms are equivalent when running interpreted code.

[2] Explorer is a trademark of Texas Instruments Inc.

4.2 Extensions

4.2.1 User Interface

Since the purpose of this project was to provide a *minimal* framework for developing a full blown L simulator, many features were neglected or omitted. Most prominent among these is the barely existent user interface. Not only should this include the rote of standard software interface facilities, it should also include assorted interactive capabilities for single stepping. Ideas in this area include offering a graphic representation of the current chunk, along with detailed mnemonic displays of the processor state that will constantly evolve throughout the simulation. Thus, for instance, one could see the current instruction highlighted in a graphic representation of the currently executing code chunk. Not only does this hold aesthetic promise, it also would convey a more graphic revelation of the key bottlenecks in chunk based execution.

4.2.2 Omitted Functionality

Aside from issues of interface, various features of the L language proper have been neglected or entirely omitted. First among these is **prerequisite management**. At the date of publish, the details of how these prerequisite management features should be achieved were largely unexplored. Also, **locks and keys** were not explored. These will be the focus of the next stage of the Simulator development.

Finally, as noted at the opening, the current Simulator does only single processor (one state chunk) simulation. Since by far the most intriguing *fortès* of L involve its ability to achieve parallelism, both the multiprocessor and the multi-tasking varieties, with minimal overhead,[3] it is vital to provide simulation of multiple concurrent state chunks. It is here where most of the important concepts behind L will be explored. However, not to fret: simulating multiple concurrent states is easily achieved by, for instance, allowing the user to pass to the Simulator a list of state chunks to multi-task. Naïvely, the simulator could then simulate one step on each processor in round-robin fashion, relying on the prerequisites and lock/key mechanisms (once their implemented) for inter-process communication. Exploring more efficient techniques for communicating processes will be an ambitious project. Presumably, these explorations could be effected in the Simulator itself, suggesting that some low level

[3]Indeed, this was the motivation for introducing the chunk based computational model

communication could be hardware supported. At any rate, the current implementation should prove a sufficiently flexible system for such investigations.

Chapter 5

Summary

To summarize, we have presented a COMMON LISP based simulator for the L Architecture. Although not a full implementation of L, it does nonetheless fulfill its goals of being a flexible, respectably efficient medium for simulation. Future development of the Simulator will include exploring alternatives for implementing *prerequisite management* and *locks and keys*. Also, one vital next step for making the existing implementation more usable is to provide a *real* user interface. Other new ideas are being considered as well, in which the Simulator will presumably play a central role both in considering alternatives and in evaluating relative efficiency. Further, since an L Compiler is a logical next step in the evolution of the L Project, the Simulator should serve as a useful operational semantic to the compiler designer. Finally, as a tantalizing side-point, we have every intention of reducing L to the hardware and μcode level, at which point the Explorer[1] LISP machine will rightfully be considered an Explorer **L** machine. It is our hope and aspiration that this development will prove more efficient than the current Explorer system, making a strong case for chunk based computation while offering an improvement to the Explorer system as well.

[1] Explorer is a trademark of Texas Instruments Inc.

Appendix A

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                                                                                     ;;;;;
;;;;;   THE L SIMULATOR  ---    Michael R. Blair        Rev.0           1986-05-09        ;;;;;
;;;;;                                                                                     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar *current-state-chunk* (new-state-chunk))

;;;;
;;;;
;;;;    CHUNK MANIPULATION SUBROUTINES
;;;;
;;;;

;; Grab the next code byte from the instruction stream, resolving chunk-byte references
;;  ... may cross a code chunk boundary.

(defsubst next-istream-byte!()
  (let ((code-chunk-ptr    (PChunk *current-state-chunk*))
        (code-chunk-offset (PCOff  *current-state-chunk*)))

      ;; At any (aligned) istream byte, we may stumble across the continuation
      ;;  pointer ... this entails an implicit jump to the continuation of the code.

      (if (ref-p code-chunk-ptr (ldb (byte 3 2) code-chunk-offset))
          (progn (setf (PChunk *current-state-chunk*)
                       (chunk-elt code-chunk-ptr (ldb (byte 3 2) code-chunk-offset)))
                 (setf (PCOff  *current-state-chunk*) 0)))

      ;; NOTE: pointers to pointers will trap to the error handler by chunk-byte()
      (let ((code-chunk-offset (PCOff  *current-state-chunk*)))
        (prog1 (chunk-byte (PChunk *current-state-chunk*) code-chunk-offset)
               (let ((postinc-uPC (1+ code-chunk-offset)))
                 ;; check for falling off the end of the chunk
                 (if (> postinc-uPC #b11111)
                     (error 'L-RUNTIME-ERROR
                            "NEXT-ISTREAM-BYTE!: postinc(uPC) chunk-boundary overflow"))
                 (setf (PCOff *current-state-chunk*) postinc-uPC))))))


;;; Dispatch table for L instruction opcodes:

(defvar *L-instruction-dispatch-table* (make-array 256))

;;;
;;; next-step!: single step the L machine by dispatching on the current opcode.
;;;

(defsubst next-step!()
  (let ((opcode (next-istream-byte!)))
    (funcall (aref *L-instruction-dispatch-table* opcode) opcode)))
```

```
;;;;
;;;;
;;;;    MAIN LOOP:  Simulates execution of the *current-state-chunk*
;;;;
;;;;

(defun LSim(&key (debug? NIL) (stream standard-output))
  (activate-state-chunk *current-state-chunk*)
  (let ((L-instruction-count 0))
    (if debug?
        (loop until (eql SUSPENDED (L-Status *current-state-chunk*))
              do (let ((result (next-step!) ))
                   (incf L-instruction-count)
                   (format stream "~&*current-state-chunk*: ")
                   (LSim-print-chunk *current-state-chunk* stream)
                   (format stream "~&L-Instruction #~D: RESULT = ~A~%" L-instruction-count
                                                                         result)
                   (read-line)))
        (loop until (eql SUSPENDED (L-Status *current-state-chunk*))
              do (progn (incf L-instruction-count) (next-step!))))
    (format stream "~2&>>>>>>><<<<<<<~%>>>  DONE  <<<~%>>>>>>><<<<<<<")
    (format stream "~2% *current-state-chunk* = ")
    (LSim-print-chunk *current-state-chunk*)
    (format stream "~%  ... in ~D L-instructions~2%" L-instruction-count))
  (values))                                    ; Don't return nuttin'
```

```
;;;;
;;;;
;;;;    SOURCE OPERANDS
;;;;
;;;;

;;;
;;; Source addressing mode handlers
;;;

(defvar *current-L-sourcecode* 0)

;;; (Sn): nth element of current state
;;;
;;;   CODE: <mode-encoding> + n  for n<8  [1 byte]

(defun source-Sn(encoding)
  (chunk-elt *current-state-chunk* (logand encoding 7)))

;;; (Sn i): ith element of chunk referenced by nth element of current state
;;;
;;;   CODE: <mode-encoding> + 8*i + n  for  i<9, n<8   [1 byte]

(defun source-Sn-i(encoding)
  (chunk-elt (chunk-elt *current-state-chunk* (logand encoding 7))
             (1- (lsh encoding -3))))

;;; (S <n> <i>): <i>th element of chunk referenced by <n>th element of current state
;;;               where  <n> is a constant n or (byte Sn i) which must resolve to a constant < 8
;;;                      <i> is a constant i or (byte Sm j) which must resolve to a constant < 9
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <n>, <i>   [4 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <n> and <i> to be arbitrary source modes, but they
;;;               must both resolve according to the above stated constraints else the error
;;;               handler will be invoked by chunk-elt.

(defun source-S-n-i()
  (let ((chunk-ref (chunk-elt *current-state-chunk* (source-operand!))))
    (chunk-elt chunk-ref (source-operand!))))

;;; (byte <chunk-ref> <index>): <index>th byte within the chunk referenced by <chunk-ref>
;;;                                  where  <chunk-ref> may be Sn or (byte Sn i)
;;;                                         <offset>    may be  j or (byte Sm j)
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <chunk ref>, <index>   [3 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <chunk-ref> and <offset> to be arbitrary source modes.
;;;               However, <chunk-ref> must resolve to a chunk-ref and <offset> must resolve
;;;               to a constant else the error handler will be invoked.
;;;               [See code for chunk-byte and chunk-elt].

(defun source-byte-c-i()
  (let ((chunk-index (source-operand!)))
    (chunk-byte (chunk-elt *current-state-chunk* chunk-index)
                (source-operand!))))
```

```
;;; (byte Sn i): ith byte of chunk referenced by Sn. n=8 for current state chunk.
;;;
;;;   CODE: <mode-encoding>, 16*i + n   [2 bytes]

(defun source-byte-Sn-i()
  (let* ((current-code-byte (next-istream-byte!))
         (n (logand current-code-byte #xF)))
    (chunk-byte (if (= n 8)                     ; n=8 means current state chunk
                    *current-state-chunk*
                    (chunk-elt *current-state-chunk* n))        ; else Sn
                (lsh current-code-byte -4))))

;;; S: current state
;;;
;;;   CODE: <mode-encoding>   [1 byte]

(defun source-S() *current-state-chunk*)

;;; k: medium (2 byte) constant
;;;
;;;   CODE: <mode-encoding>, klo, khi   [3 bytes]

(defun source-2-byte-k()
  (+ (next-istream-byte!) (* 256 (next-istream-byte!))))

;;; k  for  -64 <= k < 64: 7 bit 2's complement constant.
;;;
;;;   CODE: 128 + k   [1 byte]

(defun encode-source-1-byte-k()
  (setq *current-L-sourcecode* 128)
  (loop for i from 0 below 128
        do (eval `(defsourcecode
                    ,(if (> i 63)                               ; Negative 2's complement?
                         (- (logand #x3F (1+ (lognot i))))      ; 1's complement, increment,
                                                                ;    mask, and invert
                         i)))))

;;;
;;;  Generate uniform sourcecode names and Intern these standardized names
;;;     for the dispatch table
;;;

(defun L-sourcecode-name(n)
  (intern (format NIL "L-SOURCECODE-~2,'0X" n) 'L))

;;; postincrement(*current-L-sourcecode*)

(defun next-L-sourcecode!()
  (let ((it *current-L-sourcecode*))
    (incf *current-L-sourcecode*)
    it))

;;; defsourcecode(BODY) ==> (defun L-SOURCECODE-xx (sourcecode) BODY)
;;;                                where  xx = postinc(*current-L-sourcecode*)

(defmacro defsourcecode(&body BODY)
  `(defun ,(L-sourcecode-name (next-L-sourcecode!)) (sourcecode) (ignore sourcecode) ,@BODY))

;;; defsourcecodes(startcode no-of-encodings-plus-1 BODY)
;;;      Encodes BODY as N different sourcecodes starting at startcode
;;;          where N := (-1+ no-of-encodings-plus-1) ... i.e. no-of-encodings

(defsubst defsourcecodes(startcode no-of-encodings-plus-1 BODY)
  (setq *current-L-sourcecode* startcode)
  (let ((last-code-plus-1 (+ startcode no-of-encodings-plus-1)))
    (loop while (< *current-L-sourcecode* last-code-plus-1)
          do (eval `(defsourcecode (,BODY ,*current-L-sourcecode*))))))
```

```
;;;
;;; Now establish span of various sourcecodes
;;;

(defsubst encode-source-Sn(startcode)
  (defsourcecodes startcode 8 #'source-Sn))

(defsubst encode-source-Sn-i(startcode)
  (defsourcecodes startcode 80 #'source-Sn-i))

;;;
;;; Source operand encodings
;;;

(defun Setup-sourcecode-encodings()

  (encode-source-Sn 0)                          ; SOURCECODES 00-07
  (encode-source-Sn-i 8)                        ; SOURCECODES 08-79 (#x08-#x4F)

  (compiler-let ((*current-L-sourcecode* 123)) nil ; compiler evals first form twice, so pad
    (defsourcecode (source-S-n-i))              ; SOURCECODE 123 (#x7B)
    (defsourcecode (source-byte-c-i))           ; SOURCECODE 124 (#x7C)
    (defsourcecode (source-byte-Sn-i))          ; SOURCECODE 125 (#x7D)
    (defsourcecode (source-S))                  ; SOURCECODE 126 (#x7E)
    (defsourcecode (source-2-byte-k)) )         ; SOURCECODE 127 (#x7F)

  (encode-source-1-byte-k)                      ; SOURCECODES 128-256 (#x80 - #xFF)

  (values))

;;;
;;; Source operand dispatch table construction
;;;

;;; Dispatch table for L source addressing mode encodings:

(defvar *L-sourcecode-dispatch-table* (make-array 256))

;;; Handle an illegal sourcecode:

(defun L-sourcecode-illegal(sourcecode)
  (error 'L-RUNTIME-ERROR "Illegal L source operand code: #x~X" sourcecode))

;;; Call this table whenever the definition of any L sourcecode handler

(defun Setup-L-sourcecode-dispatch-table()
  (loop for i from 0 below 256
        do (let ((it (L-sourcecode-name i)))
             (setf (aref *L-sourcecode-dispatch-table* i)
                   (if (fboundp it) (fsymeval it)
                       #'L-sourcecode-illegal)))))

;;;
;;; Dispatch on source operand encoding
;;;

(defsubst source-operand!()
  (let ((sourcecode (next-istream-byte!)))
    (funcall (aref *L-sourcecode-dispatch-table* sourcecode) sourcecode)))
```

```
;;;;
;;;;
;;;;    DESTINATION OPERANDS
;;;;
;;;;

(defvar *current-L-destcode* 0)

;;;
;;; Destination addressing mode handlers
;;;

(defconstant SLOT-DEST   T)                     ; To be used as boolean flags
(defconstant BYTE-DEST NIL)                     ; ...  see code for store-dest()

;;; Following N destination codes are analogous to the source encodings.
;;; Note that these routines return multiple values, from which storedest()
;;;  can construct the destination address.

;;; (Sn): nth element of current state.
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding> + n  for  n<8   [1 byte]

(defun dest-Sn(encoding)
  (values SLOT-DEST *current-state-chunk* (logand encoding 7)))

;;; (Sn i): ith element of chunk referenced by nth element of current state.
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding> + 8*i + n  for  i<9, n<8   [1 byte]

(defun dest-Sn-i(encoding)
  (values SLOT-DEST (chunk-elt *current-state-chunk* (logand encoding 7))
                    (1- (lsh encoding -3))))

;;; (S <n> <i>): <i>th element of chunk referenced by <n>th element of current state
;;;               where  <n> is a constant n or (byte Sn i) which must resolve to a constant < 8
;;;                      <i> is a constant i or (byte Sm j) which must resolve to a constant < 9
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <n>, <i>   [4 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <n> and <i> to be arbitrarily defined modes, but they
;;;               must both resolve according to the above stated constraints else the error
;;;               handler will be invoked by chunk-elt.

(defun dest-S-n-i()
  (let ((chunk-ref (chunk-elt *current-state-chunk* (source-operand!))))
    (values SLOT-DEST chunk-ref (source-operand!))))

;;; (byte <chunk-ref> <index>): <index>th byte within the chunk referenced by <chunk-ref>
;;;                                    where  <chunk-ref> may be Sn or (byte Sn i)
;;;                                           <offset>    may be  j or (byte Sm j)
;;;         Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, <chunk ref>, <index>   [3 to 5 bytes]
;;;
;;; ** NOTE ** This implementation allows <chunk-ref> and <offset> to be arbitrary dest modes.
;;;               However, <chunk-ref> must resolve to a chunk-ref and <offset> must resolve
;;;               to a constant else the error handler will be invoked.
;;;               [See code for chunk-byte and chunk-elt].

(defun dest-byte-c-i()
  (let ((chunk-index (source-operand!)))
    (values BYTE-DEST
            (chunk-elt *current-state-chunk* chunk-index)
            (source-operand!))))
```

```
;;; (byte Sn i): ith byte of chunk referenced by Sn. n=8 for current state chunk.
;;;        Prerequisites unchanged.
;;;
;;;   CODE: <mode-encoding>, 16*i + n   [2 bytes]

(defun dest-byte-Sn-i()
  (let* ((current-code-byte (next-istream-byte!))
         (n (logand current-code-byte #xF)))
    (values BYTE-DEST
            (if (= n 8)                 ; n=8 means current state chunk
                *current-state-chunk*
                (chunk-elt *current-state-chunk* n))    ; else Sn
            (lsh current-code-byte -4))))

;;; (R j Sn): nth element of current state.
;;;        Sets Sn as READ-ONLY prerequisite j.
;;;
;;;   CODE: <mode-encoding> + 8*j + n   for  j<4, n<8   [1 byte]

(defun dest-R-j-Sn(encoding)
  (let ((target-slot (ldb (byte 3 0) encoding)))
    (prog1 (values SLOT-DEST encoding target-slot)
           (setf (RPreq-bit *current-state-chunk* target-slot) (ldb (byte 2 3) encoding)))))

;;; (W j Sn): nth element of current state.
;;;        Sets Sn as READ/WRITE prerequisite j.
;;;
;;;   CODE: <mode-encoding> + 8*j + n   for  j<4, n<8   [1 byte]

(defun dest-W-j-Sn(encoding)
  (let ((target-slot (ldb (byte 3 0) encoding)))
    (prog1 (values SLOT-DEST encoding target-slot)
           (setf (WPreq-bit *current-state-chunk* target-slot) (ldb (byte 2 3) encoding)))))

;;; (N j Sn): nth element of current state.
;;;        CLEARS prerequisite j.
;;;
;;;   CODE: <mode-encoding> + 8*j + n   for  j<4, n<8   [1 byte]

(defun dest-N-j-Sn(encoding)
  (let ((target-slot (ldb (byte 3 0) encoding)))
    (prog1 (values SLOT-DEST encoding target-slot)
           (setf (RPreq-bit *current-state-chunk* target-slot) 0)
           (setf (WPreq-bit *current-state-chunk* target-slot) 0))))
```

```
;;;
;;;  Generate uniform destcode names and Intern these standardized names
;;;     for the dispatch table
;;;

(defun L-destcode-name(n)
  (intern (format NIL "L-DESTCODE-~2,'0X" n) 'L))

;;; postincrement(*current-L-destcode*)

(defun next-L-destcode!()
  (let ((it *current-L-destcode*))
    (incf *current-L-destcode*)
    it))

;;; defdestcode(BODY) ==> (defun L-DESTCODE-xx (destcode) BODY)
;;;       where  xx = postinc(*current-L-destcode*)

(defmacro defdestcode(&body BODY)
  `(defun ,(L-destcode-name (next-L-destcode!)) (destcode) (ignore destcode) ,@BODY))

;;; defdestcodes(startcode no-of-encodings-plus-1 BODY)
;;;      Encodes BODY as N different destcodes starting at startcode
;;;          where N := (-1+ no-of-encodings-plus-1) ... i.e. no-of-encodings

(defsubst defdestcodes(startcode no-of-encodings-plus-1 BODY)
  (setq *current-L-destcode* startcode)
  (let ((last-code-plus-1 (+ startcode no-of-encodings-plus-1)))
    (loop while (< *current-L-destcode* last-code-plus-1)
          do (eval `(defdestcode (,BODY ,*current-L-destcode*))))))

;;;
;;; Now establish span of various sourcecodes
;;;

(defsubst encode-dest-Sn(startcode)
  (defdestcodes startcode 8 #'dest-Sn))

(defsubst encode-dest-Sn-i(startcode)
  (defdestcodes startcode 80 #'dest-Sn-i))

(defsubst encode-dest-R-j-Sn(startcode)
  (defdestcodes startcode 32 #'dest-R-j-Sn))

(defsubst encode-dest-W-j-Sn(startcode)
  (defdestcodes startcode 32 #'dest-W-j-Sn))

(defun encode-dest-N-j-Sn(startcode)
  (defdestcodes startcode 32 #'dest-N-j-Sn))


;;;
;;; Destination operand encodings
;;;

(defun Setup-destcode-encodings()
  (encode-dest-Sn 0)                          ; DESTCODES 00-07
  (encode-dest-Sn-i 8)                        ; DESTCODES 08-79 (#x08-#x4F)

  (compiler-let ((*current-L-destcode* 123)) nil ; compiler evals first form twice, so pad
    (defdestcode (dest-S-n-i))                ; DESTCODE 123 (#x7B)
    (defdestcode (dest-byte-c-i))             ; DESTCODE 124 (#x7C)
    (defdestcode (dest-byte-Sn-i)) )          ; DESTCODE 125 (#x7D)

  (encode-dest-R-j-Sn 128)                    ; DESTCODES 128-159 (#x80-#x9F)
  (encode-dest-W-j-Sn 160)                    ; DESTCODES 160-191 (#xA0-#xBF)
  (encode-dest-N-j-Sn 192)                    ; DESTCODES 192-256 (#xC0-#xDF)

  (values))
```

```
;;;
;;; Destination operand dispatch table construction
;;;

;;; Dispatch table for L destination addressing mode encodings:

(defvar *L-destcode-dispatch-table* (make-array 256))

;;; Handle an illegal destcode:

(defun L-destcode-illegal(destcode)
  (error 'L-RUNTIME-ERROR "Illegal L destination operand code: #x~X" destcode))

;;; Call this table whenever the definition of any L destcode addressing mode handler

(defun Setup-L-destcode-dispatch-table()
  (loop for i from 0 below 256
        do (let ((it (L-destcode-name i)))
             (setf (aref *L-destcode-dispatch-table* i)
                   (if (fboundp it) (fsymeval it)
                       #'L-destcode-illegal)))))

;;;
;;; Destination operand dispatch
;;;

(defsubst dest-operand!()
  (let ((destcode (next-istream-byte!)))
    (funcall (aref *L-destcode-dispatch-table* destcode) destcode)))
```

```lisp
;;;;
;;;;
;;;;  L INSTRUCTION SET OPCODES
;;;;
;;;;

(defvar *current-L-opcode* 0)

;;;
;;; Instruction opcode handlers and their opcode encodings
;;;

;;; Nullary opcode

(defun NOP() 'NOP)                                    ; NOP ... returns 'NOP

;;; Arithmetic and Logic Instructions

(defun L-add(x y)                             ; ADD: 32-bit 2's complement ... sets CCs
  ;; V - Overflow bit: arguments of same sign, but result different ... toggle sign of result
  ;; N - Negative bit ... toggled upon overflow
  ;; C - Carry bit: carry from Most Significant Bit
  ;; Z - Zero bit
  (let* ((sign-x (logand #x80000000 x))
         (sum (+ x y))
         (result (logand #xFFFFFFFF sum))
         (sign-result (logand #x80000000 result))
         (CCs (+ (if (zerop result) Z-BIT 0)
                 (if (zerop sign-result) 0 N-bit)
                 (if (zerop (logand #x100000000 sum)) 0 C-BIT))))

    (if (and (eq sign-x (logand #x80000000 y))          ; arguments have like signs ...
             (not (eq sign-result sign-x)))             ; ... but differ from result's sign
        (set-CCs (logxor CCs (+ N-BIT V-BIT)))          ; ... so force V and toggle N-bit
        (set-CCs CCs))                                  ; Install the CC bits
    result))

(defun L-sub(x y)                             ; SUBTRACT: 32-bit 2's complement ... sets CCs
  ;; V - Overflow bit: args' signs differ as do signs of x and result ... toggle sign of result
  ;; N - Negative bit ... toggled upon overflow
  ;; C - Carry bit
  ;; Z - Zero bit
  (let* ((sign-x (logand #x80000000 x))
         (x-same-sign-as-y? (eq sign-x (logand #x80000000 y)))
         (diff (- x y))
         (result (logand diff #xFFFFFFFF))
         (sign-result (logand #x80000000 result))
         (carry (logand #x100000000 diff))
         (x-and-result-signs-differ? (not (eq sign-result sign-x)))
         (CCs (+ (if (zerop result) Z-BIT 0)
                 (if (zerop sign-result) 0 N-bit)
                 carry)))

    (if (and x-same-sign-as-y?                  ; arguments have like signs ...
             x-and-result-signs-differ?)        ; ... but differ from result's sign ==> C
        (setq CCs (+ CCs C-BIT)))               ; ... so force a C bit
    (if (and (not x-same-sign-as-y?)            ; arguments have different signs ...
             x-and-result-signs-differ?)        ; ... and signs of x and result differ
        (set-CCs (logxor CCs (+ N-BIT V-BIT)))  ; ... so force V and toggle N
        (set-CCs CCs))                          ; Install the CC bits
    result))

(defun L-move(x) (L-add x 0))                   ; return the source operand, setting CCs.

(defun L-neg(x)                                 ; NEGATE: 32-bit 2's complement ... sets CCs
  (L-sub 0 x))                                  ; Special case of L-SUB
```

```
;;; Compare: x-y ... set CCs only (i.e. no destination)

(defun L-compare()
  (let* ((x (source-operand!))
         (y (source-operand!)))
    (L-sub x y)))               ; No store-dest.


;;; Logic Opcodes ... set CCs

(defun L-and() #'logand)                        ; Use some LISP primitives ...
(defun L-or () #'logior)

(defun L-not(x) (L-add (lognot x) 0))           ; 1's complement


;;; Unconditional Jumps (the long and short of it)

;;; Jump opcodes are followed by a jump-code, where a jump-code is of the format:
;;;
;;;     branch/jump     jump-code     side-effect(s)
;;;     -----------     ---------     --------------
;;;
;;;       Branch:        000sssbb      uPC := sssbb
;;;
;;;         Jump:        PPPsssbb       PC := contents of slot PPP of current code chunk
;;;                    (PPP ~= 000)    uPC := sssbb
;;;
;;;
;;;     ... Clearly then, a branch is just a short jump, a jump is just a long branch,
;;;     and a full continuation is a hop, skip and a jump.
;;;    (And a horse is a horse, of course, of course.)

(defun L-jmp()
  (let ((jump-code-byte (next-istream-byte!)))
    (let ((branch-if-zero-else-target-slot (ldb (byte 3 5) jump-code-byte))
          (target-uPC (mask-field (byte 5 0) jump-code-byte)))
      (if (not (zerop branch-if-zero-else-target-slot)) ; jump?
          (setf (PChunk *current-state-chunk*)
                (chunk-elt (PChunk *current-state-chunk*) branch-if-zero-else-target-slot)))
      (setf (PCOff *current-state-chunk*) target-uPC)   ; branch

      ;; Informative return value for single stepping ...
      (if (zerop branch-if-zero-else-target-slot)
          'BRANCH
          (if (zerop (PCOff *current-state-chunk*))
              'JUMP
              'JUMP&BRANCH)))))

;;; Conditional Jumps ... simply test condition bits then do L-jmp if satisfied,
;;;                          else skip next istream-byte.

(defsubst jump-iff-0(bits-set?)
  (if bits-set?
      (progn (next-istream-byte!) 'SKIP)        ; If not zero, punt the jump (i.e. skip)
      (L-jmp)))                                 ; Else jump

(defsubst jump-iff-not-0(bits-set?)
  (if bits-set?
      (L-jmp)                                   ; If not zero, then jump
      (progn (next-istream-byte!) 'SKIP)))      ; Else punt the jump (i.e. skip)



(defmacro defcondition(jump-condition cc-mask)
  `(,jump-condition (ldb-test (byte 6 0)
                              (logand ,cc-mask (L-CCs *current-state-chunk*)))))
```

```
(defun le () (defcondition jump-iff-not-0 Z-BIT))                          ; je  = jv
(defun lne() (defcondition jump-iff-0     Z-BIT))                          ; jne = jnv

(defun jle() (defcondition jump-iff-not-0 (logior Z-BIT N-BIT)))           ; jle
(defun lgt() (defcondition jump-iff-0     (logior Z-BIT N-BIT)))           ; jgt

(defun jlt() (defcondition jump-iff-not-0 N-BIT))                          ; jlt = jn
(defun jge() (defcondition jump-iff-0     N-BIT))                          ; jge = jnn

(defun jlos() (defcondition jump-iff-not-0 (logior Z-BIT C-BIT)))          ; jlos
(defun lhi () (defcondition jump-iff-0     (logior Z-BIT C-BIT)))          ; jhi

(defun jlo () (defcondition jump-iff-not-0 C-BIT))                         ; jlo  = jc
(defun jhis() (defcondition jump-iff-0     C-BIT))                         ; jhis = jnc

(defun jz () (defcondition jump-iff-not-0 Z-BIT))                          ; jz
(defun jnz() (defcondition jump-iff-0     Z-BIT))                          ; jnz

(defun jr () (defcondition jump-iff-not-0 R-BIT))                          ; jr
(defun jnr() (defcondition jump-iff-0     R-BIT))                          ; jnr


;;; Procedure Linkage

(defun L-fork()
  (let ((new-state (new-state-chunk)))
    (setf (ReturnLink new-state) *current-state-chunk*)
    (setf (PChunk new-state) (PChunk *current-state-chunk*))    ; to access the next istream byte
    (setf (PCOff new-state) (PCOff *current-state-chunk*))      ; for the jump ...
    (next-istream-byte!) ; return to byte past the jump code byte
    (setq *current-state-chunk* new-state)
    (L-jmp)
    (activate-state-chunk *current-state-chunk*))
  'FORK)

(defun L-call()
  (suspend-state-chunk *current-state-chunk*)
  (L-fork)
  'CALL)

(defun L-return()
  (let ((return-state-chunk (ReturnLink *current-state-chunk*)))
    ( suspend-state-chunk *current-state-chunk*)
    (activate-state-chunk return-state-chunk)
    (setq *current-state-chunk* return-state-chunk))
  'RETURN)


;;; Thread Control

(defun L-wakeup () (activate-state-chunk (source-operand!)) 'WAKEUP)
(defun L-suspend() ( suspend-state-chunk (source-operand!)) 'SUSPEND)

(defun L-cocall()
  (suspend-state-chunk *current-state-chunk*)
  (L-wakeup)
  'COCALL)

(defun L-thread()
  (L-move (new-state-chunk))
  'THREAD)
```

```
;;; Locks and Keys

(defun L-lock  () (print "UNLOCK: scaffolding") 'UNLOCK)
(defun L-unlock() (print "UNLOCK: scaffolding") 'UNLOCK)


;;; Prerequisite Management

(defun L-preq   () (print   "PREQ: scaffolding")    'PREQ)
(defun L-nopreq () (print "NOPREQ: scaffolding")  'NOPREQ)
(defun L-clrpreq() (print "NOPREQ: scaffolding") 'CLRPREQ)


;;;
;;;  Generate uniform opcode names and Intern these standardized names
;;;     for the dispatch table
;;;

(defun L-opcode-name(n)
  (intern (format NIL "L-OPCODE-~2,'0X" n) 'L))

;;; postincrement(*current-L-opcode*)

(defun next-L-opcode!()
  (let ((it *current-L-opcode*))
    (incf *current-L-opcode*)
    it))

;;; definstruction(BODY) ==> (defun L-OPCODE-xx (opcode) BODY)
;;;         where  xx = postinc(*current-L-opcode*)

(defmacro definstruction(&body BODY)
  `(defun ,(L-opcode-name (next-L-opcode!)) (opcode) (ignore opcode) ,@BODY))


;;;
;;; To save keystrokes, exploit some uniformity among arity classes ...
;;;

;;; store-dest: store the result bits into the specified dest-operand address

(defsubst store-dest(bits)
  (multiple-value-bind (slot? c i) (dest-operand!)
    (if slot? (setf (chunk-elt  c i) bits)
              (prog1 (setf (chunk-byte c i) bits)
                     (if (ref-p c i) (set-CCs R-BIT))))
    bits))                                          ; a nice return value

;;; Unary opcode template

(defmacro unop(f)
  `(let ((result (,f (source-operand!))))
     (store-dest result)))

;;; Binary opcode template

(defmacro binop(f)
  `(let* ((x (source-operand!))
          (y (source-operand!)))
     (store-dest (,f x y))))

;;; Logical opcode template

(defmacro logical(f)
  `(let* ((x (source-operand!))
          (y (source-operand!)))
     (store-dest (L-add (,f x y) 0))))
```

```
;;;
;;; Instruction opcode encodings
;;;

;; Opcodes  0-19 for general arithmetic operations
;;         20-29    logical operations
;;         30-49    jumps on compares (combinations of CC bits)
;;         50-59    call/return mechanisms
;;         60-69    cocall, threading, sleeping
;;         70-99    for future use
;;       100-109    locking and prereqs
;;       110-255    for future use (deadlock breaking, surrendering locks, abort, etc.

(defun Setup-instruction-encodings()

  (compiler-let ((*current-L-opcode* 0)) nil    ; compiler evals first form twice, so pad

    ;; No Operation

    (definstruction (NOP))                      ; OPCODE 00: NOP ... returns 'NOP

    ;; Arithmetic and Logic Instructions

    (definstruction (binop L-add))              ; OPCODE 01: ADD ... sets CCs
    (definstruction (binop L-sub)) )            ; OPCODE 02: SUB ... sets CCs

  (compiler-let ((*current-L-opcode* 10)) nil   ; compiler evals first form twice, so pad
    (definstruction (unop L-move)) )            ; OPCODE 10 (#x0A): MOVE ... data transfer


  (compiler-let ((*current-L-opcode* 20)) nil   ; compiler evals first form twice, so pad

    (definstruction (unop L-neg))               ; OPCODE 20 (#x14): NEG ... 2's comp ... sets CCs
    (definstruction (L-compare))                ; OPCODE 21 (#x15): CMP ... compare ... sets CCs

    ;; Logic Opcodes

    (definstruction (logical L-and))            ; OPCODE 22 (#x16): AND
    (definstruction (logical L-ior))            ; OPCODE 23 (#x17):  OR ... inclusive
    (definstruction (unop L-not)) )             ; OPCODE 24 (#x18): NOT ... 1's complement


  (compiler-let ((*current-L-opcode* 30)) nil ; compiler evals first form twice, so pad

    ;; Unconditional Jump(s) ... Branch and/or Jump

    (definstruction (L-jmp))                    ; OPCODE 30 (#x1E): JMP ... Branch and/or Jump

    ;; Conditional Jumps

    (definstruction (je))                       ; OPCODE 31 (#x1F): JE  = JV
    (definstruction (jne))                      ; OPCODE 32 (#x20): JNE = JNV

    (definstruction (jle))                      ; OPCODE 33 (#x21): JLE
    (definstruction (jgt))                      ; OPCODE 34 (#x22): JGT

    (definstruction (jlt))                      ; OPCODE 35 (#x23): JLT = JN
    (definstruction (jge))                      ; OPCODE 36 (#x24): JGE = JNN

    (definstruction (jlos))                     ; OPCODE 37 (#x25): JLOS
    (definstruction (jhi))                      ; OPCODE 38 (#x26): JHI

    (definstruction (jlo))                      ; OPCODE 39 (#x27): JLO  = JC
    (definstruction (jhis))                     ; OPCODE 40 (#x28): JHIS = JNC

    (definstruction (jz))                       ; OPCODE 41 (#x29): JZ
    (definstruction (jnz))                      ; OPCODE 42 (#x2A): JNZ

    (definstruction (jr))                       ; OPCODE 43 (#x2B): JR
    (definstruction (jnr)) )                    ; OPCODE 44 (#x2C): JNR
```

```
  (compiler-let ((*current-L-opcode* 50)) nil ; compiler evals first form twice, so pad

    ;; Procedure Linkage

    (definstruction (L-call))                  ; OPCODE 50 (#x32): CALL
    (definstruction (L-fork))                  ; OPCODE 51 (#x33): FORK
    (definstruction (L-return)) )              ; OPCODE 52 (#x34): RETURN


  (compiler-let ((*current-L-opcode* 60)) nil ; compiler evals first form twice, so pad

    ;; Thread Control

    (definstruction (L-cocall))                ; OPCODE 60 (#x3C): COCALL
    (definstruction (L-thread))                ; OPCODE 61 (#x3D): THREAD
    (definstruction (L-wakeup))                ; OPCODE 62 (#x3E): WAKEUP
    (definstruction (L-suspend)) )             ; OPCODE 63 (#x3F): SUSPEND


  (compiler-let ((*current-L-opcode* 100)) nil ; compiler evals first form twice, so pad

    ;; Locks and Keys

    (definstruction (L-lock))                  ; OPCODE 100 (#x64): LOCK
    (definstruction (L-unlock))                ; OPCODE 101 (#x65): UNLOCK

    ;; Prerequisite Management

    (definstruction (L-preq))                  ; OPCODE 102 (#x66): PREQ
    (definstruction (L-nopreq))                ; OPCODE 103 (#x67): NOPREQ
    (definstruction (L-clrpreq)) )             ; OPCODE 104 (#x68): CLRPREQ

  (values))

;;;
;;; Instruction opcode dispatch table construction
;;;

;;; Handle an illegal opcode:

(defun L-opcode-illegal(opcode)
  (error 'L-RUNTIME-ERROR "Illegal L opcode: ~D (#x~X)" opcode opcode))

;;; Call this table whenever the definition of any L instruction handler

(defun Setup-L-instruction-dispatch-table()
  (loop for i from 0 below 256
        do (let ((it (L-opcode-name i)))
             (setf (aref *L-instruction-dispatch-table* i)
                   (if (fboundp it) (fsymeval it)
                       #'L-opcode-illegal)))))
```

```
;;;;
;;;;
;;;; MISCELLANEOUS UTILITY FUNCTIONS
;;;;
;;;;

;;;
;;;  Set up the SIMULATOR STATE
;;;

(defun Setup-L-simulator()
  (format T     "~&Instruction encodings ............")
  (Setup-instruction-encodings)
  (format T "DONE~%Source mode encodings ............")
  (Setup-sourcecode-encodings)
  (format T "DONE~%Destination mode encodings ........")
  (Setup-destcode-encodings)
  (format T "DONE~%Instruction dispatch table ........")
  (Setup-L-instruction-dispatch-table)
  (format T "DONE~%Source mode dispatch table ........")
  (Setup-L-sourcecode-dispatch-table)
  (format T "DONE~%Destination mode dispatch table ...")
  (Setup-L-destcode-dispatch-table)
  (format T "DONE~%")
  'DONE)

(defun Setup() (Setup-L-simulator))            ; Shorthand

;;;
;;; A bit more informative version of PRINT-CHUNK()
;;;

(defun LSim-print-chunk(c &optional (stream standard-output))
  (tyo #\{ stream)
  (loop for i from 0 below Chunk-Elts
        do (if (zerop (aref (Chunk-refbits c) i))
               (format stream "#x~8,'0X" (chunk-elt c i))
               (format stream "<~A>" (substring (format nil "~A" c) 29 37)))
        (if (< i (- chunk-Elts 1)) (tyo #\space stream)))
  (format stream "}~%")
  (values))
```

Appendix B

```
;;; Common lisp implementation of L chunks, very preliminary  9/85 Steve Ward
;;;                          slightly modified and expanded  5/86 Ziggy
;;;
;;;        * note that accessing forms can be used in setf
;;;             -- eg, (setf (chunk-elt c 3) 4) works fine.
;;;        * In general, LISP objects passed as chunk elements are presumed to
;;;             be either integers or chunks; ref bits are set accordingly.
;;;
;;;    (new-chunk type)                           ; make a new chunk.
;;;    (chunk-elt c i)                            ; return ith element of chunk c.
;;;    (ref-p c i)                                ; is ith element of c a ref?
;;;
;;;    (chunk-type c)                             ; type element (= element 8) of c
;;;
;;;    (assuming-chunk c body)                    ; ERROR iff c isn't a chunk.
;;;
;;;    (assuming-type c type body)                ; ERROR iff c isn't a chunk
;;;                                               ;   of type "type"
;;;
;;;    (assuming-state c body)                    ; ERROR iff c isn't a state chunk.
;;;
;;;    (PChunk state)                             ; code chunk element of state
;;;    (PCOff state)                              ; code offset of state

(defconstant Chunk-Elts     9)                    ; Total elements
(defconstant Chunk-Type-Elt 8)                    ; Element # of type elt

;;; Some scalar types:

(defconstant L-State-Type    #xF0000001)          ; computation state
(defconstant L-Code-Type     #xF0000002)          ; pure code

;;; As a temporary kludge, we represent a chunk using two arrays for the
;;;   elements.  The ith element is stored in (aref scalers i) iff its
;;;   a scalar (zerop (aref refbits i)) or in (aref refs i) iff its a
;;;   chunk reference.
;;; All of these details are shielded from code, which accesses chunks
;;;   through the access functions defined subsequently.

(defstruct (Chunk)
   (scalars (make-array Chunk-Elts
                              :type 'art-32b)) ; Array type: 32 bit
   (refs    (make-array Chunk-Elts
                              ))
   (refbits (make-array Chunk-Elts
                              :element-type '(mod 2)))
   bytes
   )
```

```
; Access to the Type element of a chunk:
(defsubst Chunk-type(c)(Chunk-ref-scalar c chunk-Type-Elt))

;;; Is ith element of chunk c a ref?
(defsubst ref-p(c i)
   (not (zerop (aref (Chunk-refbits c) i))))

;;; Access to ith element of a chunk: also divided up into ref and scalar access

(defsubst chunk-elt(c n)
   (if (ref-p c n)
       (aref (Chunk-refs c) n)
       (aref (Chunk-scalars c) n)))

(defsetf chunk-elt(c n) (value)
   `(if (eq 'chunk (type-of ,value))
        (progn (setf (aref (Chunk-refs ,c) ,n) ,value)
                 (setf (aref (chunk-refbits ,c) ,n) 1)
                 ,value)
        (progn (setf (aref (chunk-refs ,c) ,n) nil)         ; Avoid extraneous refs.
                 (setf (aref (chunk-scalars ,c) ,n) ,value)
                 (setf (aref (chunk-refbits ,c) ,n) 0)
                 ,value)))


(defsubst chunk-ref-reference(c n)
   (if (ref-p c n)
       (aref (Chunk-refs c) n)
       (error 'L-RUNTIME-ERROR "Chunk ~A slot ~A is a scalar" c n)))

(defsetf chunk-ref-reference(c n) (value)
     `(if (eq 'chunk (type-of ,value))
          (progn (setf (aref (Chunk-refs ,c) ,n) ,value)
                 (setf (aref (chunk-refbits ,c) ,n) 1)
                 ,value)))


(defsubst chunk-ref-scalar(c n)
   (if (ref-p c n)
       (error 'L-RUNTIME-ERROR "Chunk ~A slot ~A is a reference" c n)
       (aref (Chunk-scalars c) n)))

(defsetf chunk-ref-scalar (c n) (value)
  `(if (eq 'chunk (type-of ,value))
       (error 'L-RUNTIME-ERROR "Chunk elt is a scalar")
       (progn (setf (aref (chunk-refs ,c) ,n) nil)        ; Avoid extraneous refs.
              (setf (aref (chunk-scalars ,c) ,n) ,value)
              (setf (aref (chunk-refbits ,c) ,n) 0)
              ,value)))
```

```lisp
;;; Following functions allow access to consecutive bytes of a chunk,
;;;   presuming 4 (8-bit) bytes to a scalar.
(defsubst chunk-byte(c n)
   (let ((i (lsh n -2)))
;       (if (>= n 32)(error 'L-RUNTIME-ERROR "Chunk-byte index (~D) too big" n))
      (if (ref-p c i)(error 'L-RUNTIME-ERROR "Chunk-byte ~D indexes ref (~D)" n i))
      (aref (chunk-bytes c) n)))


(defsetf chunk-byte(c n)(value)
   `(let ((chunk-byte-index (lsh ,n -2)))
      (if (ref-p ,c chunk-byte-index)
          (setf (chunk-ref-reference ,c chunk-byte-index) 0))
      (setf (aref (chunk-bytes ,c) ,n) ,value)))


;;; Make a new chunk (make-chunk won't set bytes right... STUPID LISP!
(defun new-chunk(&optional (type 0)
                       &aux (it (make-chunk)))
   (setf (chunk-bytes it)
         (make-array (* 4 Chunk-Elts)
                     :element-type '(mod 256)
                     :displaced-to (Chunk-scalars it)))
   (setf (chunk-type it) type)
   it)

;;; Make a state chunk ... special since newly created states should be SUSPENDed
(defsubst new-state-chunk(&optional (status SUSPENDED))
  (let ((state-chunk (new-chunk L-State-Type)))
    (setf (L-Status state-chunk) status)
    state-chunk))                              ; return the new state-chunk


;;; Simple chunk-printing function, mostly for debugging:
(defun print-chunk(c &optional (stream *standard-output*))
  (format stream "{")
  (loop for i from 0 below Chunk-Elts
        do (if (zerop (aref (Chunk-refbits c) i))
               (format stream "~A" (chunk-elt c i))
               (print-chunk (chunk-elt c i) stream))
           (if (< i (- Chunk-Elts 1))(format stream " " )) )
  (format stream "}"))

;;; Some type-checking macros.

(defmacro Assuming-chunk (c &rest BODY)
   `(if (not (eql (type-of ,c) 'CHUNK))
        (error 'L-RUNTIME-ERROR "~S Not a chunk~&" ,c)
        ,@BODY))

(defmacro Assuming-type(c type &rest BODY)
   `(Assuming-chunk ,c (if (not (eql (chunk-type ,c) ,type))
                           (error 'L-RUNTIME-ERROR
                                         "Bad type ~S, expected L chunk type ~S"
                                          (chunk-type ,c) ,type)
                           ,@BODY)))

(defmacro Assuming-State(c &rest BODY)
   `(Assuming-chunk ,c (if (not (eql (chunk-type ,c) L-State-Type))
                           (error 'L-RUNTIME-ERROR
                                       "Bad type ~S, expected L-State-Type ~S"
                                        (chunk-type ,c) L-State-Type)
                            ,@BODY)))
```

```
;;; Some utilities for dealing with STATE chunks:

; State slot allocation and access

(defconstant     uSTATE-SLOT 0)
(defconstant         PC-SLOT 1)
(defconstant RETURNLINK-SLOT 2)

; Access the micro state:
(defsubst uState(State) (chunk-elt State uSTATE-SLOT))

; Access the code chunk pointer:
(defsubst PChunk(State) (chunk-elt State PC-SLOT))

; Access the return linkage pointer:
(defsubst ReturnLink(State) (chunk-elt State RETURNLINK-SLOT))


;;; Some details about the micro-state element contents:
;;;
;;; BYTE:  [3]       [2]       [1]                [0]
;;;         _______   _______   _______________    ___________
;;;       | WPreq | | RPreq | | RunStat | CCs |  | xxx | uPC |
;;;         -------   -------   ---------------    -----------
;;; # Bits:   8         8           2      6        3     5


; uState word allocation ... note that setf works just fine with ldb.
(defsubst PCOff    (State) (ldb (byte 5 0)  (uState State)))    ; First 5 bits
(defsubst L-CCs    (State) (ldb (byte 6 8)  (uState State)))    ; Next 6 bits
(defsubst L-Status (State) (ldb (byte 2 14) (uState State)))    ;      2 bits
(defsubst RPreq    (State) (ldb (byte 8 16) (uState State)))    ;      8 bits
(defsubst WPreq    (State) (ldb (byte 8 24) (uState State)))    ;      8 bits

;;; Some specific bit allocations

; Condition bit assignments

(defconstant R-BIT #b10000)
(defconstant V-BIT #b01000)
(defconstant N-BIT #b00100)
(defconstant C-BIT #b00010)
(defconstant Z-BIT #b00001)

(defsubst set-CCs(ccode) (setf (L-CCs *current-state-chunk*) ccode))

; Run status bit assignments

(defconstant RUNNABLE  #b11)
(defconstant SUSPENDED #b10)
(defconstant INACTIVE  #b00)

(defsubst    suspend-state-chunk(c) (setf (L-Status c) SUSPENDED))
(defsubst   activate-state-chunk(c) (setf (L-Status c) RUNNABLE))
(defsubst deactivate-state-chunk(c) (setf (L-Status c) INACTIVE))

; Access the prereq bit for slot s.
(defsubst RPreq-bit(State s) (ldb (byte 1 s) (RPreq State)))
(defsubst WPreq-bit(State s) (ldb (byte 1 s) (WPreq State)))
```

Appendix C

```
;;;;
;;;;
;;;;  Some sample code for testing Fibonacci(n) as presented in [4].
;;;;
;;;;

; Opcodes  0-19 for general arithmetic operations
;         20-29     logical operations
;         30-49     jumps on compares (combinations of CC bits)
;         50-59     call/return mechanisms
;         60-69     cocall, threading, sleeping
;         70-99     for future use
;       100-109     locking and prereqs
;       110-255     for future use (deadlock breaking, surrendering locks, abort, etc.

(defvar *caller-bits*
;       '( call    PC 7    suspend  S           ; slot 0
;          0       0       0        0           ; slot 1
;          0       0       0        0           ; slot 2
;          0       0       0        0           ; slot 3
;          0       0       0        0           ; slot 4
;          0       0       0        0           ; slot 5
;          0       0       0        0           ; slot 6
;          << ref to fib code chunk >>          ; slot 7
;          )
      '( 50          #b11100000 63          126 ; slot 0 - NOTE jump
         0           0          0           0   ; slot 1
         0           0          0           0   ; slot 2
         0           0          0           0   ; slot 3
         0           0          0           0   ; slot 4
         0           0          0           0   ; slot 5
         0           0          0           0   ; slot 6
         0           0          0           0   ; slot 7
         ))

(defvar *caller-code-chunk*)

(defun make-caller-code-chunk(callee-code-chunk)
  (let ((caller-code (new-chunk L-Code-Type)))
    (loop for i from 0 below 32
          do (setf (chunk-byte caller-code i) (elt *caller-bits* i)))
    (setf (chunk-ref-reference caller-code 7) callee-code-chunk)
    (setq *caller-code-chunk* caller-code))
  (format T "~&*caller-code-chunk*:  ")
  (LSim-print-chunk *caller-code-chunk*))
```

```
(defvar *caller-state-chunk*)

(defun make-caller-state-chunk(n)
  (let ((caller-state (new-chunk L-State-Type)))
    (setf (PChunk caller-state) *caller-code-chunk*)    ; Install PC
    (setf (ReturnLink caller-state) caller-state)       ; KLUDGE! Make caller-state return to itself
    (setf (chunk-elt caller-state 4) n)                 ; Install argument in slot 4
    (setq *caller-state-chunk* caller-state))
  (activate-state-chunk *caller-state-chunk*)
  (format T "~&*caller-state-chunk*: ")
  (LSim-print-chunk *caller-state-chunk*))


(defvar *fib-bits*
;       '( cmp     RTN S4  #2      jlt                  ; slot 0
;          6.0     sub     RTN S4  #1                   ; slot 1
;          S4      call    PC 7    move                 ; slot 2
;          S4      S5      sub     RTN S4               ; slot 3
;          #2      S4      call    0.0                  ; slot 4
;          add     S4      S5      RTN S4               ; slot 5
;          return  0       0       0                    ; slot 6
;          << ref to fib code chunk >>                  ; slot 7
;          )
      '( 21          #b0101010  130         35          ; slot 0
         #b00011000  2          #b0101010   129         ; slot 1
         4           50         #b11100000  10          ; slot 2 - NOTE jump
         4           5          2           #b0101010   ; slot 3
         130         4          50          #b00000000  ; slot 4 - NOTE branch
         1           4          5           #b0101010   ; slot 5
         52          0          0           0           ; slot 6
         0           0          0           0           ; slot 7
         ))

(defvar *fib-code-chunk*)

(defun make-fib-code-chunk()
  (let ((fib-code (new-chunk L-Code-Type)))
    (loop for i from 0 below 32
          do (setf (chunk-byte fib-code i) (elt *fib-bits* i)))
    (setf (chunk-ref-reference fib-code 7) fib-code)
    (setq *fib-code-chunk* fib-code))
  (format T "~&*fib-code-chunk*:     ")
  (LSim-print-chunk *fib-code-chunk*))

;;; Setup the environment to simulate Fib(n)

(defun fib(n)
  (make-fib-code-chunk)
  (make-caller-code-chunk *fib-code-chunk*)
  (make-caller-state-chunk n)
  (setq *current-state-chunk* *caller-state-chunk*)
  (values))                                     ; Don't return nuttin'


;;; Setup the environment to run code chunk ch with argument n.

(defun setup-state(c n)
  (make-caller-code-chunk c)
  (make-caller-state-chunk n)
  (setq *current-state-chunk* *caller-state-chunk*)
  (values))                                     ; Don't return nuttin'
```

Bibliography

[1] Abelson, Harold and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., 1985.

[2] DIGITAL's Sales Support Literature Group. *PDP11 Processor Handbook.* Digital Equipment Corporation, 1978.

[3] Steele, Guy L. *Common LISP: The Language.* Digital Press, Digital Equipment Corporation, 1984.

[4] Terman, Chris and Stephen A. Ward. *RTS Group – The L Project: A Proposal.* Unpublished memo of the M.I.T. Laboratory for Computer Science, Cambridge, Mass., 1986.

[5] Texas Instruments Data Systems Group. *Explorer System Software.* Texas Instruments Incorporated, Austen, Texas, 1985.

[6] Ward, Stephen A. and Robert H. Halstead, Jr. *Computation Structures.* Unpublished lecture notes for MIT course 6.004 Computation Structures. Stephen A. Ward and Robert H. Halstead, 1985.

[7] Ward, Stephen A. *L Instruction Set: Straw Man Proposal.* Unpublished memo of the M.I.T. Laboratory for Computer Science, Cambridge, Mass., 1986.